Hierarchical Planning for Long-Horizon Multi-Agent Collective Construction

Shambhavi Singh[1], Zejian Huang[1], Akshaya Kesarimangalam Srinivasan[1],
Geordan Gutow[1], Bhaskar Vundurthy[1], and Howie Choset[1]

Abstract— We develop a planner that directs robots to construct a 3D target structure composed of blocks. The robots themselves are cubes of the same size as the blocks, and they may place, carry, or remove one block at a time. When moving, robots are also allowed to climb or descend a block. A construction plan may thus build a staircase-like scaffolding of blocks to reach other blocks at higher levels. The order of block placement is important; for example, a block that sits atop other blocks must be placed after the blocks below it, and a block that needs scaffolding cannot be placed until after the scaffolding is. Prior works focus on end-to-end approaches that simultaneously plan for block placement order and inter-robot collisions. Larger structures are either intractable or yield high-cost solutions. A prior approach mitigates this by decomposing the structure into smaller components that can be planned for independently, but the computational challenge remains. We present a hierarchical approach that first 1) uses A* to determine a sequence of block placements and removals while ignoring inter-robot collision, then 2) identifies ordering constraints between block placement and removal actions, and finally (3) computes collision-free paths for multiple robots to perform said actions. Compared to an optimization approach that minimizes the number of timesteps to complete the structure, we observe a 100x reduction in computation time for comparable solutions.

I. INTRODUCTION

We envision a future where robots will assemble large structures in remote locations on Earth or even in space. This will likely require coordinating large numbers of robots. A challenge in such construction planning is that robot actions alter the world. When the structure changes, the actions available to the robots do too. Robots must therefore plan far in advance to adapt to the evolving structure, and avoid conflicts and deadlocks. To get started in studying problems with this feature, this work, along with others [1]–[5], considers a simplified problem referred to as the Multi-Agent Collective Construction (MACC) problem. In the MACC problem, cubic robots coordinate in a 3D grid world to assemble a structure made of cubic blocks (Fig. 1). Blocks and robots are the same size and occupy one cubic location in the grid. Robots can modify the environment by block pick-up or place actions. Because robots can climb or descend at most one block height at a time, they may construct temporary staircase-like scaffolding using extra blocks to make higher altitudes reachable (Fig. 2). The problem is inspired by the TERMES project [1], and initial solutions were developed in [2]–[4].
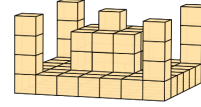
Fig. 1. An example structure for the multi-agent construction problem

A major challenge in the MACC problem is that the length of the action sequence to build a structure increases quickly as the structure gets bigger. For example, a two-block tower requires placing the two tower blocks as well as placing and removing one scaffolding block (4 pick-up/place actions total), while a tower of height three needs three scaffolding blocks (9 pick-up/place actions), and a tower of height 4 needs 16 pick-up/place actions.

As a result, MACC planning involves long time horizons. Long-horizon problems are particularly difficult as the number of valid sequences of actions usually grows exponentially with the planning horizon. Breaking planning into easier sub-problems has been shown to mitigate the computational costs of long horizon planning in other problem domains. One method is to plan for multiple smaller short-horizon tasks that can be solved quickly [5]–[10]. Alternatively, hierarchical approaches like [11]–[13] plan with abstract actions for which the horizon is shorter, then refine the solutions to include collision avoidance and other constraints.

We propose a hierarchical approach to the MACC problem, using pick-up and place actions as the abstract actions. Our approach (Fig. 3) finds solutions for structures that were too large for a prior optimal algorithm [4] yet maintains comparable solution quality on structures that [4] could solve. We formalize the MACC problem in section III. In section IV we find a single-agent plan of abstracted actions, where each action corresponds to a non-unique set of primitive action sequences that place or remove a specific block. We identify sets of abstract actions for parallel execution of this single-agent abstract plan in section V. For each set, we then present a guarantee that a multi-agent primitive action path exists. The multi-agent primitive path planning is solved using conflict-based search [14] in section VI. Finally in section VII we compare the performance of our hierarchical approach with four existing approaches [2]–[5].

II. RELATED WORKS

A. Hierarchical Planning

Hierarchical planning approaches leverage domain knowledge to identify abstract tasks that combine multiple prim-
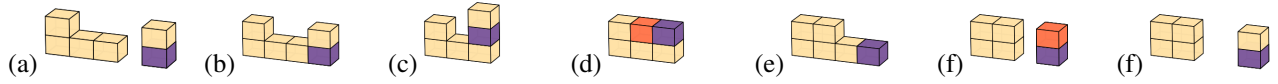
Fig. 2. (a-c) A robot (Purple) moves to an access location, (d) places a block (Orange), (e) moves back to the next action's access location, (f) picks up a different block (Orange) and (g) returns while carrying the block. The primitive actions in this sequence are: Move → Move → Move → Place → Move → Pick-up → Move.

itive actions. They generate a plan composed of such tasks and then refine it to generate the primitive action plan. For multi-arm assembly, [11] generates a roadmap for each robot, solves a relaxed problem on the roadmap using Mixed Integer Linear Programming, and then finds collision-free motion plans for each robot. The relaxed abstract plans usually require fewer timesteps than the primitive plans, which keeps the initial search tractable. The procedure for self-assembling a structure in [15] first computes a partial order plan that is enforced via run-time constraints. Primitive plans are obtained via local controllers. The method successfully plans for some large problem instances. A deep reinforcement learning method in [16] solves a multi-agent warehouse automation problem using a two-level hierarchy of a single-agent Markov Decision Process on the upper level and a multi-agent Markov game on the lower level.

Refining the abstract plan to a primitive plan can itself be computationally expensive. [17] introduces a heuristic approach to estimate the number of modifications required in the abstract plan before generating a primitive plan. These estimates are based on task decomposition graphs containing all the abstract tasks' decomposition in the planning domain.

B. Multi-Agent Collective Construction (MACC)

The TERMES [1] project developed small robots to co-operatively build a structure composed of blocks similar to the robots in size. The structures were much larger than the robots, requiring them to construct traversable paths on the structure. Inspired by the TERMES project, [2] formulated the blockworld construction task considered in this work and found that state-of-the-art domain-independent planners (as of 2014) could not solve even small instances. Therefore, they presented a heuristic single agent planner. A follow-on work [18] modified the heuristic planner for multiple agents. Both techniques plan extremely fast but require numerous primitive actions to construct the structure. The optimization-based approach in [4] improved on solution quality by formulating a sequence of Mixed Integer Linear Programs (MILPs). This minimizes the number of time steps to build the structure (the makespan), and for fixed makespan, minimizes the number of primitive actions used. While optimal, this approach is intractable for larger structures.

To improve on computation time, [5] suggests decomposing the desired structure into substructures, planning independently using the MILP approach of [4], then aggregating the solutions. For sparse and scattered structures, decomposition achieves better solution quality than earlier suboptimal approaches [2], [3] and smaller computation time than the optimal MILP approach [4]. The reliance on structural composition is however less effective for taller or densely built structures. The current work thus fits into a gap in the literature, providing high-quality construction plans for multiple agents on structures otherwise accessible only to heuristic approaches like [2].

III. PROBLEM FORMULATION

The MACC problem tasks a team of agents to construct a given 3D structure in a gravity-constrained world. The agents are mobile robots that can pick-up/place blocks, and carry blocks on top as they move. We represent the workspace as a 3D grid world, where each cubic cell of the workspace is a triplet (x, y, z), entries ranging from 1 to M. The agents and blocks both occupy one cell each. An agent may execute any of the following **primitive actions** (Fig. 2):

- **Move:** Move one step to an adjacent cell in 4 cardinal directions, climbing or descending at most one block.
- **Pick-up/Place:** Pick up or place a block at a neighboring location as long as the agent and the neighboring location are at the same height.
- **Enter/Exit:** Enter the workspace to a boundary location (may choose to be carrying a block or not), or exit the workspace from a boundary location.

Agents can place blocks on the ground (height 0) or atop another block in the workspace. Additionally, agents cannot place blocks at boundary locations, and all boundary locations remain at height 0. We assume a large depot of blocks outside of the workspace is accessible to agents through any boundary location. Since agents can climb/descend at most one block, they may place extra scaffolding blocks to access higher levels of the workspace, and remove scaffolding later.

Definition 1. We define a block that is a part of the goal structure as a **target** block and a temporarily placed block that is not part of the goal structure as a **scaffolding** block.

A problem instance is **completed** when all target blocks have been placed, all scaffolding blocks removed, and all agents have exited the workspace. We now define our metrics for evaluation:

1. **Makespan**: The total number of timesteps required to complete the problem instance,
2. **Sum of costs**: Total number of actions taken by agents,
3. **Computation Time**: The time it takes to find a solution for completing a structure.

IV. ABSTRACT-LEVEL PLANNING

An abstract action adds or removes a block from the world. This corresponds to the set of sequences of primitive actions
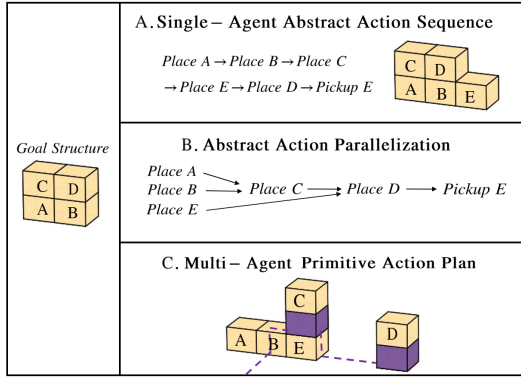
Fig. 3. For a goal structure, a single-agent sequence of block placement and removal actions is found using A* (A). This abstract action sequence is converted into a dependency graph, where directed edges indicate one action must precede another (B). Finally, conflict-based search is used to plan paths for multiple robots to build the structure (C).

such that an agent places or picks up the block and returns to a depot location. The abstract planning problem is formulated as a graph search, where a node represents a state of the world , and an edge represents an abstract action. We use A* search to find the shortest path from the empty world to the goal structure with a uniform edge cost, thereby minimizing the total abstract actions.

Definition 2. For a block location (x_b, y_b, z_b), all cardinally adjacent block locations (x_a, y_a, z_a) at the same height are called **access locations** (In Fig.2(d) robot places a block standing on an access location).

$$|x_a - x_b| + |y_a - y_b| = 1, \ z_a = z_b$$

Definition 3. A pick-up or place action is **executable** if a path exists for an agent to traverse from outside the workspace to an access location, execute the action, and return to outside the workspace.

At each search node, the available edges correspond to the executable (Definition 3) abstract actions at that state. The A* search is accelerated by the use of an informative cost-to-go heuristic and two node-ordering heuristics.

Remark 1. *The abstract action corresponding to a node n is executable in the state produced by executing the actions of each node along a path from the empty world to n.*

The cost-to-go heuristic estimates the amount of scaffolding needed to build the structure using Algorithm 1. For some node, let T' be the set of unplaced target blocks, P be the set of placed scaffold blocks, and P' be the set of scaffold blocks yet to be placed. A good estimate of the cost-to-go is $h = |T'| + |P| + 2 \cdot |P'|$; one for each unplaced structure block, one for each temporary scaffold yet to be removed, and two for each necessary scaffold yet to be placed and removed. This underestimates the number of high-level actions needed to finish the structure. However, computing $|P'|$ is challenging. Instead, we use Algorithm 1 to find an accurate under-estimate of $|P'|$ at each search node.

Algorithm 1 Scaffold Estimation

Input: A state of a search node
Output: Estimation of scaffolds required c

1: $c \leftarrow 0, \ u(x', y', z') \leftarrow 0, \forall (x', y', z') \in [1...M]^3$
2: **for** unplaced target block group g_z at height $z > 1$ **do**
3:      **for** $i = 1, 2, ..., z$ **do**
4:         $S(g_z, i) \leftarrow$ Set of scaffolds (Definition 5)
5:         **if** no target block or placed scaffold in $S(g_z, i)$ **then**
6:            $u(x', y', z') \leftarrow u(x', y', z') + 1, \forall (x', y', z') \in S(g_z, i)$
7:            $I(g_z, i) \leftarrow 1$
8: **for** unplaced target block group g_z at height $z > 1$ **do**
9:      $c \leftarrow c + \sum_{i=1}^{z-1} I(g_z, i) \cdot \min_{(x', y', z') \in S(g_z, i)} \frac{1}{u(x', y', z')}$

Definition 4. The set of blocks present at any (x, y) grid location is collectively referred to as a **tower**. The height of the tower is the number of blocks present at that location.

To construct a target tower of height z, blocks within the tower need to be placed in increasing order of height. To place the target block at height $z > 1$ when the target block at height $z - 1$ is placed, we require all scaffolds needed to place the block at $z - 1$, as well as 1 extra scaffold at height $z - 1$ 1 Manhattan distance away, 1 extra scaffold at height $z - 2$ 2 Manhattan distance away, etc., until reaching the ground. Further, if we connect adjacent target blocks of the same height into a group, the same observation holds: the whole group requires 1 extra scaffold at height $z - 1$ of 1 Manhattan distance away from the group, 1 extra scaffold at height $z - 2$ of 2 Manhattan distance away, etc. Let g_z denote a group of adjacent, unplaced target blocks at height z, and g_{xy} be the set of xy locations.

Definition 5. For each $i \in [1, z-1]$, define the **scaffold set** of $S(g_z, i)$ as the set of possible scaffolds at height $z - i$ of i Manhattan away from the group:

$$S(g_z, i) = \{(x', y', z') : (x', y') \notin g_{xy} \ \& \ z - z' = i \ \& \\ \exists (x, y) \in g_{xy} \quad |x - x'| + |y - y'| = i.\}$$

Algorithm 1 provides an underestimate of the scaffolding blocks required, at any given state of an A* search node. In Lines 2-4, for each unplaced target block group g_z, we first compute $S(g_z, i)$. Next, in Line 5, we check if g_z contains neither target blocks, nor already placed scaffolding, and if True, in Lines 6-7, we thus increment a *usefulness* counter $u(x', y', z')$ for all scaffolds within $S(g_z, i)$ and set a binary indicator $I(g_z, i)$ to 1 to indicate a scaffold is required among them. The usefulness counter of a scaffold overestimates the number of unplaced target block groups that require it. The inverse of a usefulness counter is thus an underestimate of the fraction of this scaffold required by each target group. In Lines 8-9, we compute c as the sum of the minimum of the inverse of all usefulness counters of blocks in $S(g_z, i)$ for every g_z, i whose indicator is 1.

The underlying symmetry of the structures causes many nodes to have the same *estimated* total cost to reach the goal. Two node-ordering heuristics are used to break ties. First,

Algorithm 2 Building an Action Dependency Graph

Input: Action Sequence A, Set of all scaffolding graphs
Output: Action Dependency Graph

1: $ADG \leftarrow [empty\ graph]$
2: **for** action a_i in A **do**
3:    $Loc_{a_i} \leftarrow$ 3D location of action a_i
4:    $S_{a_i} \leftarrow$ scaffolding set for Loc_{a_i}
5:    **for** a_j in $\{a_0 \rightarrow a_1 \rightarrow ... a_{i-1}\}$ **do**
6:      $Loc_{a_j} \leftarrow$ 3D location of action a_j
7:      $S_{a_j} \leftarrow$ scaffolding set for Loc_{a_j}
8:      **if** Loc_{a_i} is a descendent of Loc_{a_j} in S_{a_i} **then**
9:        $ADG \leftarrow ADG$ with edge from a_j to a_i
10:     **else if** $xy(Loc_{a_i}) = xy(Loc_{a_j})$ **then**
11:       $ADG \leftarrow ADG$ with edge from a_j to a_i
12:     **else if** a_i removes block in S_{a_j} **or** a_i places block on block in S_{a_j} **then**
13:       $ADG \leftarrow ADG$ with edge from a_j to a_i

actions that place a target block are explored before actions that place or remove scaffolding. Second, we compute for new scaffolding blocks the number of towers they can support (whose scaffolding set they fall in) and prioritize actions that support more towers.

V. ABSTRACT ACTION PARALLELIZATION

A* produces a sequence $A = \{a_0 \rightarrow a_1 \rightarrow \cdots \rightarrow a_{n-1}\}$ of abstract actions that a single agent can execute. Many actions in A could be executed simultaneously, but some cannot. We say that a dependency exists between any such pair of abstract actions. Let $a_c \in A$, a_p that precedes a_c in A, and a_s succeed a_c in A. In general, an action a_c can both depend on and be a dependency of multiple actions.

For example, all block placement actions that affect the same tower must occur in the same order as in the abstract action sequence. This ensures an *action location* is available.

Property 1: a_c is dependent on a_p if both actions occur at the same x and y coordinates and a_p precedes a_c in A.

Similarly, if a_p places a scaffolding block for another action a_c on a *tower*, a_p must occur before a_c to ensure the availability of an *access location* and scaffolding for a_c.

Property 2: a_c is dependent on a_p, if the block manipulated by a_p falls within the scaffolding set (Definition 5) of the *tower* for a_c, $S_c = \cup_{i=1}^{z-1} S(\{(x_c, y_c, z_c)\}, i)$ where (x_c, y_c, z_c) is the block location of a_c.

Suppose an action's sole *access location* is atop a scaffolding block. The pick-up of that scaffolding block must occur after the *access location* is no longer required.

Property 3: a_c is dependent on a_p if a_c either removes a block in the scaffolding set $S_p = \cup_{i=1}^{l_{zp}-1} S(\{(x_p, y_p, z_p)\}, i)$ for a_p or places a block atop a block in S_p.

It is worth noting that we add the dependencies of *Property 2* and *Property 3* conservatively to avoid finding the exact scaffolding block being used. If two *access locations* exist for an abstract action, the two previous actions on both *access locations* satisfy *Property 2*, but only one is needed.

Similarly, if two subsequent actions make those *access locations* unavailable, both satisfy *Property 3* for that action.

Algorithm 2 takes A as input and generates an Action Dependency Graph (ADG). Each node in the ADG represents an abstract action from A, and the directed edges between the nodes represent dependencies between their respective actions. Since all dependencies are only added to past actions, the ADG is a directed acyclic graph.

Definition 6. Set of abstract actions that are not dependent on each other can be grouped into a **round**. Agents can be tasked to execute all abstract actions in a round parallely.

The goal of Algorithm 3 is to produce a sequence of rounds containing all actions in A, such that each round is feasible for multi-agent pathfinding provided all actions in preceding rounds have occurred. The candidate is a topological sort [19] of the ADG identical to that proposed in [20]. The first round contains the ADG nodes with no parents. Subsequent rounds are produced by removing the nodes in the previous round and selecting the nodes that no longer have parents. These rounds form the first candidate allocation P (Lines $5-11$), where $P[i]$ represents the actions in round i. For each round $P[i]$ we check if each action $a_c \in P[i]$ is executable in the $State_c$ produced by actions in preceding rounds and actions in $P[i]$ that precede a_c in A (Lines $12-16$). If not, we find an action a_j that succeeds/precedes a_c in A but preceded/succeeded it in P. Add a dependency to force a_c and a_j to occur in the order they appeared in A (Lines $17-25$), recompute P, and test again. This procedure eventually adds to each action a dependency on every action that precedes it in A, which will produce a candidate allocation identical to A. In practice, Lines $17-25$ either never get called or quickly converge to a set of feasible rounds due to the conservative nature of the dependencies that already exist between actions.

Theorem 1. *A single agent path exists for each action in round **q** given that (1) all actions in all past rounds have been executed, (2) all preceding actions in **A** in round **q** have been executed, and (3) no other actions have been executed.*

Proof. Correctness: Algorithm 3 returns a set of rounds only if it identifies (in Line 15) a single agent path for each action in the world specified by assumptions (1), (2), and (3).

Completeness: The single-agent action sequence is single-agent executable. Thus, placing a_1 in round 1, a_2 in round 2, etc., is satisfactory. Algorithm 3 eventually produces precisely these rounds, and thus always returns a solution. $\square$

Next, we provide a sufficient condition for a set of abstract actions (i.e. a round) to be solvable by a Multi-Agent Path-Finding (MAPF) [21]–[24] algorithm. Let *parking locations* be locations in the workspace where no blocks can be placed. We overload the term *well-formed* condition [23] and formally define it below, then prove this condition ensures solvability.

Definition 7. A round is **well-formed** if there exists a single-agent path to execute round's actions and a cycle of *parking*

Algorithm 3 Generate Feasible Rounds of Task Allocation

Input: Action dependency graph *ADG*, Action Sequence *A*
Output: Set of feasible rounds *P*

1: *Done* ← FALSE
2: **while** NOT *Done* **do**
3:   $ADG_{working}$ ← *ADG*
4:   Round index q ← 1
5:   **while** number of nodes in $ADG_{working}$ > 0 **do**
6:     $P[q]$ ← empty sequence
7:     *Root* ← actions that are root nodes in $ADG_{working}$
8:     **for** a_i in *Root* **do**
9:       push a_i onto $P[q]$
10:      Remove node of a_i from $ADG_{working}$
11:     q ← $q+1$
12:   Current state $State_c$ ← *Empty World*
13:   **for** round i in $\{1,2,...q-1\}$ **do**
14:     **for** a_c in $P[i]$ **do**
15:       **if** a_c is executable in $State_c$ **then**
16:         $State_c$ ← State with a_c executed in $State_c$
17:       **else**
18:         **for** round k in $\{1,2,...q-1\}$ **do**
19:           **for** action a_j in $P[k]$ **do**
20:             **if** $k \leq i$ **and** a_j succeeds a_c in A **then**
21:               ADG ← ADG with edge from a_j to a_c
22:               GOTO line 2
23:             **if** $k \geq i$ **and** a_j precedes a_c in A **then**
24:               ADG ← ADG with edge from a_c to a_j
25:               GOTO line 2
26:   *Done* ← TRUE

locations whose length is no less than the number of agents.

Lemma 1. *All well-formed rounds are solvable.*

Proof. Assume all agents are located in the cycle of parking locations at the initial timestep. Start with the agent assigned with the first abstract action in the single-agent path. Since the single-agent path is valid, there must exist a path for this agent that begins at boundary locations, finishes the block action, and returns to the boundary locations. When it needs to occupy any boundary locations, all other agents can move along the cycle to avoid collisions since the size of the cycle is at least as large as the number of agents. We repeat this process for all actions until completion. □

Theorem 2. *Every round of abstract actions is solvable.*

Proof. By Theorem 1, there exists a single-agent path to execute all abstract actions within the round. The boundary locations of the workspace constitute a cycle of parking locations as long as the workspace is large enough. The round of abstract actions is well-formed and thus solvable. □

VI. MULTI-AGENT PATH FINDING

Finding collision-free paths for all agents to execute assigned abstract actions is similar to the Multi-Agent Pickup and Delivery [23] problem, a variation of MAPF. Conflict-Based Search (CBS) [14], commonly used for solving such

problems, is complete and optimal with respect to makespan and sum of costs, and works for general graphs. It follows a bi-level approach where the low-level plans paths for each agent given constraints specified by the high-level. The high-level operates on a Conflict Tree (CT), finds conflicts between individual paths in a search node, and resolves them by adding two child nodes in CT, each with an extra constraint for an agent involved in the conflict. In a **vertex conflict** two agents occupy the same vertex at the same timestep. In an **edge conflict** two agents traverse the same edge at the same timestep. Two constraints are used to resolve these conflicts. A **vertex constraint** (r_i,x,y,t) constrains agent r_i to occupy (x,y) at time t. An **edge constraint** (r_i,x,y,x',y',t) constrains agent r_i to move along the edge $(x,y) \rightarrow (x',y')$ at time t.

Block actions can alter the workspace, changing the validity of movement and block actions. This requires two extra conflicts and one extra constraint. An **agent-block conflict** (r_i,r_j,x,y,t) happens when agent r_i occupies (x,y) at time t, while agent r_j performs a block action to (x,y) at the same time. A **height conflict** (r_i,r_j,x,y,z,t) happens when agent r_i occupies (x,y,z) at t, while z does not match the height $h_t(x,y)$ and agent r_j's block action is located at (x,y). A **block constraint** (r_i,t) prohibits agent r_i from performing its block action at time t.

At a timestep t in a low-level search, let $h_t(x,y)$ denote the height of the tower at (x,y) at this time. We add all locations $(x,y,h_t(x,y)+1)$ right above each tower to the default set of vertices for the low-level search. We then add locations that will be made available by tasked abstract actions as extra vertices, given a set of abstract actions allowed to be executed at this timestep. For instance, if location $(2,2)$ has height 0 at time 0 and a task is assigned to place down a block here, then $(2,2,1),(2,2,2)$ are both added as vertices. An edge is added between two vertices if the Manhattan distance between their xy coordinates is 1 and the difference between their z coordinates is no more than 1.

At a particular CT node, we construct a sequence of height maps based on when agents perform their block actions. This sequence is used to detect conflicts. Let t_j be the timestep when agent r_j performs its block action. The two new conflicts can be resolved by adding two sets of constraints, $C1$ and $C2$, to two children CT nodes respectively.

To solve an agent-block conflict, let $C1 = \{(r_i,x,y,t)\}$ (prohibit visiting the location) and $C2 = \{(r_j,t)\}$ (prohibit performing block action). To solve a height conflict, we must consider whether the block action has been performed when the conflict is detected. If the action has not been performed $(t < t_j)$, let $C1 = \{(r_i,x,y,z,t')|t' <= t_j\}$ and $C2 = \emptyset$. The former prohibits visiting the location until block action is performed. If the action has been performed $(t > t_j)$, let $C1 = \{(r_i,x,y,z,t')|t' > t_j)\}$ and $C2 = \{(r_j,t')|t' <= t+1\}$. The former prohibits visiting the location after block action, and the latter prohibits block action until visiting the location.

The low-level plans single-agent paths to satisfy all constraints. A path needs to contain several waypoints: 1) picking or placing from the depot if necessary, 2) executing the assigned task, and 3) returning to a boundary location.
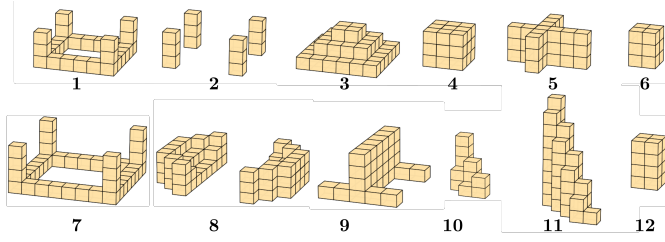
Fig. 4. (1-6) Test Structures used in baselines [3] [4] [5] (7-12) Additional, larger structures considered in this work

TABLE I

RESULTS FOR STRUCTURE 1-6 IN FIG. 4

A - Optimization Approach [4], B - Decomposition [5], C - Ours

Structure		1	2	3	4	5	6
Sum of Costs	A	173	124	-	-	-	160
	B	179	128	326	263	381	161
	C	197	138	405	355	400	166
Makespan	A	13	13	-	-	-	21
	B	17	14	44	75	90	40
	C	21	21	42	57	66	33
Solve Time(s)	A	1115.0	139.0	-	-	-	1715.2
	B	259.4	235.9	318.1	758.6	688.5	287.9
	C	1.0	0.9	5.2	5.4	14.7	0.9

We use multi-label A* (MLA*) [24] to search for the optimal path between waypoints.

VII. EXPERIMENTS AND RESULTS

We compare our approach with four existing approaches to the MACC problem. Except where noted results are reported for a 10x10x10 grid space, where each structure is run once, with a run-time limit of 10,000 seconds on a system with 16GB RAM and Intel® Core™ i7-7700K CPU @ 4.20GHz 8 processor. We implement our algorithm in Python3 and do not use multiple threads. Methods in [4], [5] solve optimization problems using Gurobi, which exploits multi-threading. In Table I, we compare our approach with the makespan optimal algorithm of [4] and the decomposition-based suboptimal approach of [5] on six test structures from [3]–[5] using at most 20 agents. Our approach is 100-1000x times faster than that of [4] and more than 40x faster than [5], with similar solution quality in sum-of-costs and makespan.

Further, we present 6 additional structures with heights higher than 3 and observe that methods [4], [5] fail to find solutions within 10,000s for these structures. In the makespan-optimal baseline, the size of the optimization

TABLE II

RESULTS FOR STRUCTURE 7-12 IN FIG. 4 FOR OUR APPROACH

Structure	7	8	9	10	11	12
Sum of Costs	286	874	312	244	645	279
Makespan	41	73	79	62	353	75
Solve Time(s)	2.8	315.4	7.6	110.7	7.4	6.3

TABLE III

RESULTS FOR RANDOM STRUCTURES IN 7x7x4 ENVIRONMENT [5]

A - Optimization Approach [4], B - Decomposition [5], C - Ours

Approach	A	B	C
Sum of Costs	83.7	54.1	102.7
Makespan	18.0	29.5	30.2
Solve Time(s)	423.5	126.0	1.2

TABLE IV

COMPARISON OF SUM OF COSTS WITH OTHER NON-OPTIMAL METHODS

Structure	1	2	3	4	5	6
Tree-based [2]	1144	836	1590	2120	2180	836
RL-based [3]	3040	1026	3056	3252	2804	1276
Ours	197	138	405	355	400	163

increases exponentially with rising makespan. Thus, these structures which require long makespans are intractable for [4]. Decomposing into smaller substructures as in [5] does not reduce the makespan of each substructure enough to make planning tractable. Our algorithm, as reported in Table II, finds a solution for each structure. Planning requires less than ten seconds except for structures 8 and 10. In structure 10, 107.9 of the 110.7 seconds are spent in the abstract action search. We suspect this is due to optimal abstract action sequences being rare for structure 10. Structure 8 (also in Fig. 1), which is larger and exists in a 12x12x10 workspace, spends 300 of 315 seconds in CBS. The abstract plan is found in only 7.5 seconds.

Table III contains comparisons with [4], [5] on randomly generated structures from [5]. On average, our approach improves run-time 100x with comparable solution quality.

Finally, in Table IV, we present a comparison of the sum-of-costs for the Heuristic-based Method [2] and the average sum-of-costs of the Distributed Reinforcement Learning-based Method [3]. Although both methods provide solutions instantaneously, the average sum of costs is an order of magnitude higher than our approach. The solutions for all reported structures are visualized in https://28shambhavi.github.io/hierarchical_macc/

VIII. CONCLUSION

In this work, we use our hierarchical planning approach to solve the MACC problem, producing solutions two orders of magnitude faster than baselines and maintaining comparable solution quality. We observe that, in the MACC problem, planning for abstract actions before computing robot actions generates significant benefits over end-to-end planning methods. In our future work, we aim to extend the planner to handle variations of the problem, such as planning for robots carrying larger blocks that enable construction of structures with canopies and hollow spaces. We also aim to integrate bounded sub-optimal methods to reduce computation time further and help plan for larger structures.

9008

References

[1] H. Durrant-Whyte, N. Roy, and P. Abbeel, *TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction*, 2012, pp. 257–264.

[2] T. Kumar, S. Jung, and S. Koenig, "A tree-based algorithm for construction robots," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 2014, pp. 481–489, 05 2014.

[3] G. Sartoretti, Y. Wu, W. Paivine, T. K. S. Kumar, S. Koenig, and H. Choset, "Distributed reinforcement learning for multi-robot decentralized collective construction," in *International Symposium on Distributed Autonomous Robotic Systems*, 2018.

[4] E. Lam, P. J. Stuckey, S. Koenig, and T. K. S. Kumar, "Exact approaches to the multi-agent collective construction problem," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 743–758.

[5] A. Kesarimangalam Srinivasan, S. Singh, G. Gutow, H. Choset, and B. Vundurthy, "Multi-agent Collective Construction using 3D Decomposition," *arXiv e-prints*, p. arXiv:2309.00985, Sept. 2023.

[6] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon control for temporal logic specifications," in *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 101–110.

[7] J. Tůmová and D. V. Dimarogonas, "A receding horizon approach to multi-agent planning from local ltl specifications," in *2014 American Control Conference*, 2014, pp. 1775–1780.

[8] V. N. Hartmann, A. Orthey, D. Driess, O. S. Oguz, and M. Toussaint, "Long-horizon multi-robot rearrangement planning for construction assembly," *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 239–252, 2023.

[9] M. Ossenkopf, G. Castro, F. Pessacg, K. Geihs, and P. De Cristóforis, "Long-horizon active slam system for multi-agent coordinated exploration," in *2019 European Conference on Mobile Robots (ECMR)*, 2019, pp. 1–6.

[10] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 13, pp. 11 272–11 281, May 2021.

[11] J. Chen, J. Li, Y. Huang, C. Garrett, D. Sun, C. Fan, A. Hofmann, C. Mueller, S. Koenig, and B. C. Williams, "Cooperative task and motion planning for multi-arm assembly systems," 2022.

[12] D. Le and E. Plaku, "Cooperative multi-robot sampling-based motion planning with dynamics," *Proceedings of the International Conference*

[20] M. C. Er, "A Parallel Computation Approach to Topological Sorting," *The Computer Journal*, vol. 26, no. 4, pp. 293–295, 11 1983. [Online]. Available: https://doi.org/10.1093/comjnl/26.4.293

on Automated Planning and Scheduling, vol. 27, no. 1, pp. 513–521, Jun. 2017.

[13] Y. Li, X.-Y. Jiao, B. Sun, Q. Zhang, and J. Yang, "Multi-welfare-robot cooperation framework for multi-task assignment in healthcare facilities based on multi-agent system," *2021 IEEE International Conference on Intelligence and Safety for Robotics (ISR)*, pp. 413–416, 2021.

[14] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," in *Artificial Intelligence*, 2012.

[15] A. Grushin and J. A. Reggia, "Automated design of distributed control rules for the self-assembly of prespecified artificial structures," *Robotics and Autonomous Systems*, vol. 56, no. 4, pp. 334–359, 2008.

[16] D. S. Carvalho and B. Sengupta, "Hierarchically structured scheduling and execution of tasks in a multi-agent environment," 2022.

[17] P. Bercher, S. Keen, and S. Biundo-Stephan, "Hybrid planning heuristics based on task decomposition graphs," *Proceedings of the International Symposium on Combinatorial Search*, 2014.

[18] T. Cai, D. Y. Zhang, T. S. Kumar, S. Koenig, and N. Ayanian, "Local search on trees and a framework for automated construction using multiple identical robots: (extended abstract)," in *Proceedings of the 2016 International Conference on Autonomous Agents amp; Multiagent Systems*, ser. AAMAS '16. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2016, p. 1301–1302.

[19] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, p. 558–562, nov 1962. [Online]. Available: https://doi.org/10.1145/368996.369025

[21] D. Silver, "Cooperative pathfinding," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 1, no. 1, pp. 117–122, Sep. 2021. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/18726

[22] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak, "Multi-agent pathfinding: Definitions, variants, and benchmarks," 2019.

[23] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Adaptive Agents and Multi-Agent Systems*, 2017.

[24] X. Zhong, J. Li, S. Koenig, and H. Ma, "Optimal and bounded-suboptimal multi-goal task assignment and path finding," *2022 International Conference on Robotics and Automation (ICRA)*, pp. 10 731–10 737, 2022.